

System and Methods for Implementing an Explicit Interface
Member in a Computer Programming Language

Copyright Notice and Permission:

5 A portion of the disclosure of this patent document may contain material that is
subject to copyright protection. The copyright owner has no objection to the facsimile
reproduction by anyone of the patent document or the patent disclosure, as it appears in the
Patent and Trademark Office patent files or records, but otherwise reserves all copyright
rights whatsoever. The following notice shall apply to this document Copyright © 2000,
10 Microsoft Corp.

Cross Reference to Related Applications:

 This application relates to U.S. Patent Appln. Nos. (Attorney Docket Nos. MSFT-571
and MSFT-572).

Field of the Invention:

 The present invention relates to the provision of an explicit interface member in
connection with a computer programming language.

Background of the Invention:

 In computing terms, a program is a specific set of ordered operations for a computer to
perform. With an elementary form of programming language known as machine language, a
programmer familiar with machine language can peek and poke data into and out from
computer memory, and perform other simple mathematical transformations on data. Over
25 time, however, the desired range of functionality of computer programs has increased quite
significantly making programming in machine language generally cumbersome. As a result,
proxy programming languages capable of being compiled into machine language and capable
of much higher levels of logic have evolved. Examples of such evolving languages include
COBOL, Fortran, Basic, Pascal, C, C++, Lisp, Visual Basic, C# and many others. Some
30 programming languages tend to be better than others at performing some types of tasks, but in
general, the later in time the programming language was introduced, the more complex

functionality that the programming language possesses empowering the developer more and more over time. Additionally, class libraries containing methods, classes and types for certain tasks are available so that, for example, a developer coding mathematical equations need not derive and implement the sine function from scratch, but need merely include and refer to the
5 mathematical library containing the sine function.

Further increasing the need for evolved software in today's computing environments is that software is being transported from computing device to computing device and across platforms more and more. Thus, developers are becoming interested in aspects of the software beyond bare bones standalone personal computer (PC) functionality. To further
10 illustrate how programming languages continue to evolve, in one of the first descriptions of a computer program by John von Neumann in 1945, a program was defined as a one-at-a-time sequence of instructions that the computer follows. Typically, the program is put into a storage area accessible to the computer. The computer gets one instruction and performs it and then gets the next instruction. The storage area or memory can also contain the data on
15 which the instruction operates. A program is also a special kind of "data" that tells how to operate on application or user data. While not incorrect for certain simple programs, the view is one based on the simplistic world of standalone computing and one focused on the functionality of the software program.

However, since that time, with the advent of parallel processing, complex computer
20 programming languages, transmission of programs and data across networks, and cross platform computing, the techniques have grown to be considerably more complex, and capable of much more than the simple standalone instruction by instruction model once known.

For more general background, programs can be characterized as interactive or batch in
25 terms of what drives them and how continuously they run. An interactive program receives data from an interactive user or possibly from another program that simulates an interactive user. A batch program runs and does its work, and then stops. Batch programs can be started by interactive users who request their interactive program to run the batch program. A command interpreter or a Web browser is an example of an interactive program. A program
30 that computes and prints out a company payroll is an example of a batch program. Print jobs are also batch programs.

When one creates a program, one writes it using some kind of computer language and the collection(s) of language statements are the source program(s). One then compiles the source program, along with any utilized libraries, with a special program called a language compiler, and the result is called an object program (not to be confused with object-oriented programming). There are several synonyms for an object program, including object module, executable program and compiled program. The object program contains the string of 0s and 1s called machine language with which the logic processor works. The machine language of the computer is constructed by the language compiler with an understanding of the computer's logic architecture, including the set of possible computer instructions and the bit length of an instruction.

Other source programs, such as dynamic link libraries (DLL) are collections of small programs, any of which can be called when needed by a larger program that is running in the computer. The small program that lets the larger program communicate with a specific device such as a printer or scanner is often packaged as a DLL program (usually referred to as a DLL file). DLL files that support specific device operation are known as device drivers. DLL files are an example of files that may be compiled at run-time.

The advantage of DLL files is that, because they don't get loaded into random access memory (RAM) together with the main program, space is saved in RAM. When and if a DLL file is needed, then it is loaded and executed. For example, as long as a user of Microsoft Word® is editing a document, the printer DLL file does not need to be loaded into RAM. If the user decides to print the document, then the Word application causes the printer DLL file to be loaded into the execution space for execution.

A DLL file is often given a ".dll" file name suffix. DLL files are dynamically linked with the program that uses them during program execution rather than being compiled with the main program. The set of such files (or the DLL) is somewhat comparable to the library routines provided with programming languages such as Fortran, Basic, Pascal, C, C++, C#, etc.

The above background illustrates (1) that computer programming needs can change quickly in a very short time along with the changing computing environments in which they are intended to operate and (2) that computing programming environments are considerably more complex than they once were. As computing environments become more and more

complex, there is generally a greater need for uniformity of functionality across platforms, uniformity among programming language editors, uniformity among programming language compilers and run time aspects of programming. In short, as today's computer system architectures have quickly expanded to the limits of the Earth via global networks, the types of programming tasks that are possible and desirable has also expanded to new limits. For example, since a program may traverse hundreds, if not hundreds of thousands of computers, as a result of copying, downloading or other transmission of the source code, developed by a plurality of unknown developers, affiliated with one another or not, there are a number of scenarios in which implementing an explicit interface member has become desirable.

While many object oriented programming languages allow for implementation of an interface, thus far, no programming language has presented an explicit interface member that is adequate for today's distributed computing environments in which, inter alia, maximum control may not be available for exercising over end user client bits. With today's programming languages, typically, a *class* or *struct* implements an interface member by providing a public instance member with the same signature. However, first, there is generally no way to control the independent development of interfaces that may ultimately conflict. Second, current programming languages allow only the implementation of public members, and thus do not allow the implementation of private interfaces, which enable greater developing flexibility. Public interfaces, for example, may ultimately place additional unintended overhead on a class definition that implements the interface. Third, present programming languages do not enable a developer to implement specific versions of a generic interface without potential conflict.

Thus, there is a need for a robust system and methods for implementing an explicit interface member in connection with a computer programming language. It would be desirable to provide a mechanism that prevents conflicts between independently developed interfaces. It would be further desirable to provide a mechanism for implementing private interface members. It would be still further desirable to provide a mechanism for implementing specific versions of generic interfaces that do not conflict.

Summary of the Invention:

In view of the foregoing, the present invention provides a system and methods for

implementing an explicit interface member in connection with a computer programming language. Thus, a mechanism is provided that prevents conflicts between independently developed interfaces, enables privately implemented interface members and enables specific versions of generic interfaces that are free from conflict.

5 Other features of the present invention are described below.

Brief Description of the Drawings:

10 The system and methods for providing an explicit interface member in connection with a programming language are further described with reference to the accompanying drawings in which:

Figure 1 is a block diagram representing an exemplary network environment in which the present invention may be implemented;

Figure 2 is a block diagram representing an exemplary non-limiting computing device that may be present in an exemplary network environment, such as described in Figure 1;

15 Figures 3A and 3B show computer programming language pseudocode illustrating an overview of the explicit interface member implementation of the present invention;

Figures 4A through 4D show computer programming language pseudocode illustrating various syntactical and other definitional aspects of explicit interface member implementations in accordance with the present invention;

20 Figures 5A through 5F show computer programming language pseudocode illustrating various aspects of and exemplary rules for interface mapping in connection with explicit interface member implementations in accordance with the present invention;

25 Figures 6A through 6E show computer programming language pseudocode illustrating interface implementation inheritance in connection with explicit interface member implementations in accordance with the present invention;

Figures 7A through 7C show computer programming language pseudocode illustrating interface re-implementation in connection with explicit interface member implementations in accordance with the present invention;

30 Figures 8A through 8F show computer programming language pseudocode illustrating the problem of implementing two independently-developed and conflicting interfaces and the solution thereof in accordance with explicit interface member implementations of the present

invention;

Figure 9 shows computer programming language pseudocode illustrating the problem of public only interfaces and the solution thereof in accordance with explicit interface member implementations of the present invention; and

Figures 10A through 10C show computer programming language pseudocode illustrating the problem of conflicting specific implementations of a generic interface and the solution thereof in accordance with explicit interface member implementations of the present invention.

Detailed Description of the Invention:

Overview of the Invention

The present invention provides a system and methods for providing explicit interface member in connection with computer programming languages. Thus, in accordance with the present invention, a mechanism is provided that prevents conflicts between independently developed interfaces. The invention further enables privately implemented interface members, as well as public implementations. The invention also enables specific implementations of generic interfaces that do not conflict.

As mentioned in the background, some support for interfaces is common in object-oriented programming languages today. Typically, a class or struct implements an interface member by providing a public instance member with the same signature. For instance, the program code 300 of Fig. 3A defines an interface *Interface1* containing a method *F* and a class *C* that implements the interface by including a public instance method *F*. However, as described in the background, such support for public interfaces alone is limited.

C# supports the above model, but in accordance with the present invention additionally supports an innovative language feature in this area, namely explicit interface member implementation. Explicit interface member implementation permits a class or struct to implement one or more interface members by explicitly specifying the relationship between the class or struct member and the interface member. This association is made by specifying the name of the class or struct member as a qualified name, consisting of the interface name and the interface member name.

For instance, the program code 310 of Fig. 3B includes a class *C* that implements the

interface *Interface1* with the method *Interface1.F*. The member *Interface1.F* is an example of an explicit interface member implementation, described in greater detail below.

Exemplary Network Environments

5 One of ordinary skill in the art can appreciate that a computer 110 or other client device can be deployed as part of a computer network. In this regard, the present invention pertains to any computer system having any number of memory or storage units, and any number of applications and processes occurring across any number of storage units or volumes. The present invention may apply to an environment with server computers and
10 client computers deployed in a network environment, having remote or local storage. The present invention may also apply to a standalone computing device, having programming language functionality, interpretation and execution capabilities.

Fig. 1 illustrates an exemplary network environment, with a server in communication with client computers via a network, in which the present invention may be employed. As
15 shown, a number of servers 10a, 10b, etc., are interconnected via a communications network 14, which may be a LAN, WAN, intranet, the Internet, etc., with a number of client or remote computing devices 110a, 110b, 110c, 110d, 110e, etc., such as a portable computer, handheld computer, thin client, networked appliance, or other device, such as a VCR, TV, and the like in accordance with the present invention. It is thus contemplated that the present invention
20 may apply to any computing device in connection with which it is desirable to perform programming services, with interfacing functionality. In a network environment in which the communications network 14 is the Internet, for example, the servers 10 can be Web servers with which the clients 110a, 110b, 110c, 110d, 110e, etc. communicate via any of a number of known protocols such as hypertext transfer protocol (HTTP). Communications may be
25 wired or wireless, where appropriate. Client devices 110 may or may not communicate via communications network 14, and may have independent communications associated therewith. For example, in the case of a TV or VCR, there may or may not be a networked aspect to the control thereof. Each client computer 110 and server computer 10 may be equipped with various application program modules 135 and with connections or access to
30 various types of storage elements or objects, across which files may be stored or to which portion(s) of files may be downloaded or migrated. Any server 10a, 10b, etc. may be

responsible for the maintenance and updating of a database 20 or other storage element in accordance with the present invention, such as a database 20 for storing software having the interfacing of the present invention. Thus, the present invention can be utilized in a computer network environment having client computers 110a, 110b, etc. for accessing and interacting with a computer network 14 and server computers 10a, 10b, etc. for interacting with client computers 110a, 110b, etc. and other devices 111 and databases 20.

Exemplary Computing Device

Fig. 2 and the following discussion are intended to provide a brief general description of a suitable computing environment in which the invention may be implemented. It should be understood, however, that handheld, portable and other computing devices of all kinds are contemplated for use in connection with the present invention. While a general purpose computer is described below, this is but one example, and such a computing device may include a browser for connecting to a network, such as the Internet. Additionally, the present invention may be implemented in an environment of networked hosted services in which very little or minimal client resources are implicated, e.g., a networked environment in which the client device serves merely as a browser or interface to the World Wide Web.

Although not required, the invention will be described in the general context of computer-executable instructions, such as program modules, being executed by one or more computers, such as client workstations, servers or other devices. Generally, program modules include routines, programs, objects, components, data structures and the like that perform particular tasks or implement particular abstract data types. Typically, the functionality of the program modules may be combined or distributed as desired in various embodiments.

Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations. Other well known computing systems, environments, and/or configurations that may be suitable for use with the invention include, but are not limited to, personal computers (PCs), automated teller machines, server computers, hand-held or laptop devices, multi-processor systems, microprocessor-based systems, programmable consumer electronics, network PCs, minicomputers, mainframe computers and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network or

other data transmission medium. In a distributed computing environment, program modules may be located in both local and remote computer storage media including memory storage devices.

Fig. 2 thus illustrates an example of a suitable computing system environment 100 in which the invention may be implemented, although as made clear above, the computing system environment 100 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the invention. Neither should the computing environment 100 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment 100.

With reference to Fig. 2, an exemplary system for implementing the invention includes a general purpose computing device in the form of a computer 110. Components of computer 110 may include, but are not limited to, a processing unit 120, a system memory 130, and a system bus 121 that couples various system components including the system memory to the processing unit 120. The system bus 121 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus (also known as Mezzanine bus).

Computer 110 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 110 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CDROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired

information and which can accessed by computer 110. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or
5 more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media.

10 The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 110, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program
15 modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, Fig. 2 illustrates operating system 134, application programs 135, other program modules 136, and program data 137.

The computer 110 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, Fig. 2 illustrates a
20 hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156, such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary
25 operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through a non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory
30 interface, such as interface 150.

The drives and their associated computer storage media discussed above and

illustrated in Fig. 2 provide storage of computer readable instructions, data structures, program modules and other data for the computer 110. In Fig. 2, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 110 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus 121, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190. In addition to the monitor, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195.

The computer 110 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 180. The remote computer 180 may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer 110, although only a memory storage device 181 has been illustrated in Fig. 2. The logical connections depicted in Fig. 2 include a local area network (LAN) 171 and a wide area network (WAN) 173, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the computer 110 is connected to the LAN 171 through a network interface or adapter 170. When used in a WAN networking environment, the computer 110 typically includes a modem 172 or other means for establishing communications over the WAN 173, such as the Internet. The modem 172,

which may be internal or external, may be connected to the system bus 121 via the user input interface 160, or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer 110, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, Fig. 2 illustrates remote application programs 185 as residing on memory device 181. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

Software may be designed using many different methods, including object-oriented programming methods. C++, Java, etc. are examples of common object-oriented programming languages that provide functionality associated with object-oriented programming. Object-oriented programming methods provide a means to encapsulate data members, e.g. variables, and member functions, e.g. methods, that operate on that data into single entity called a class. Object-oriented programming methods also provide means to create new classes based on existing classes.

An object is an instance of a class. The data members of an object are attributes that are stored inside the computer memory, and the methods are executable computer code that act upon this data, along with potentially providing other services. The present invention provides a robust system and techniques for explicitly implementing interfaces for classes and methods.

Exemplary Languages and the .NET Framework

In exemplary embodiments of the explicit interface mechanism as described herein, the present invention is described in connection with the C# programming language. However, one of ordinary skill in the art will readily recognize that the interfacing techniques of the present invention may be implemented with any programming language, such as Fortran, Pascal, Visual Basic, C, C++, Java, etc.

C# is a simple, modern, object oriented, and type-safe programming language derived from C and C++. C#, pronounced "C sharp" like the musical note, is firmly planted in the C and C++ family tree of languages, and will be familiar to programmers having an understanding of the C and C++ programming languages, and other object-oriented programming languages. Generally, C# combines the high productivity of Visual Basic and

the raw power of C++, and provides many unique programming features as well.

C# is provided as part of Microsoft Visual Studio 7.0. In addition to C#, Visual Studio supports Visual Basic, Visual C++, and the scripting languages VBScript and JScript. All of these languages provide access to the Microsoft .NET platform, which includes a common execution engine and a rich class library. The Microsoft .NET platform defines a Common Language Subset (CLS), a sort of lingua franca that ensures seamless interoperability between CLS-compliant languages and class libraries. For C# developers, this means that even though C# is a new language, it has complete access to the same rich class libraries that are used by seasoned tools such as Visual Basic and Visual C++. C# itself may not include a class library.

.Net is a computing framework that has been developed in light of the convergence of personal computing and the Internet. Individuals and business users alike are provided with a seamlessly interoperable and Web-enabled interface for applications and computing devices, making computing activities increasingly Web browser or network-oriented. In general, the .Net platform includes servers, building-block services, such as Web-based data storage and downloadable device software.

Generally speaking, the .Net platform provides (1) the ability to make the entire range of computing devices work together and to have user information automatically updated and synchronized on all of them, (2) increased interactive capability for Web sites, enabled by greater use of XML (Extensible Markup Language) rather than HTML, (3) online services that feature customized access and delivery of products and services to the user from a central starting point for the management of various applications, such as e-mail, for example, or software, such as Office .Net, (4) centralized data storage, which will increase efficiency and ease of access to information, as well as synchronization of information among users and devices, (5) the ability to integrate various communications media, such as e-mail, faxes, and telephones, (6) for developers, the ability to create reusable modules, thereby increasing productivity and reducing the number of programming errors and (7) many other cross-platform integration features as well. While exemplary embodiments herein are described in connection with C#, the interfacing of the present invention may be supported in all of Microsoft's .NET languages. Thus, as one of ordinary skill in the art can appreciate, it would be desirable to incorporate the interfacing functionality of the present invention into any programming language.

Explicit Interface Member Implementation

The present invention provides an explicit interface member implementation in connection with a programming language, such as C#. The explicit interface member implementation of the present invention permits a class or struct to implement one or more interface members by explicitly specifying the relationship between the class or struct member and the interface member. Explicit interface member implementation is a language feature that differentiates a programming language, such as C#, from all other known computer programming platforms and languages. Any language that includes interface implementation could potentially benefit by adding a feature similar to the below-described explicit interface member implementation in C#. Explicit interface member implementations

Thus, for purposes of implementing interfaces in accordance with the present invention, a class or struct may declare explicit interface member implementations. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example, in the code 400 of Fig. 4A, illustrating an exemplary explicit interface member implementation, *ICloneable.Clone* and *IComparable.CompareTo* are explicit interface member implementations.

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can only be accessed through an interface instance, and is in that case referenced simply by its member name. It is also an error for an explicit interface member implementation to include access modifiers, just as it is an error to include the *abstract*, *virtual*, *override*, or *static* modifiers.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are not accessible through their fully qualified name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense also public.

Explicit interface member implementations serve at least two purposes not provided by available prior art provision of interfacing. First, since explicit interface member implementations are not accessible through class or struct instances, they allow interface

implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of the class or struct. Second, explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the class *Shape* illustrated by code 410 of Fig. 4B, the declaration of *Comparable.CompareTo* is invalid because *Comparable* is not listed in the base class list of *Shape* and is not a base interface of *Cloneable*. Likewise, in the declarations of *Shape* and *Ellipse* in code 420 of Fig. 4C, the declaration of *Cloneable.Clone* in *Ellipse* is in error because *Cloneable* is not explicitly listed in the base class list of *Ellipse*.

The fully qualified name of an interface member references the interface in which the member was declared. Thus, in the declarations of code 430 of Fig. 4D, the explicit interface member implementation of *Paint* is written as *Control.Paint*.

A class or struct provides implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.

Interface mapping for a class or struct *C* locates an implementation for each member of each interface specified in the base class list of *C*. The implementation of a particular interface member *I.M*, where *I* is the interface in which the member *M* is declared, is determined by examining each class or struct *S*, starting with *C*. The process repeats for each successive base class of *C*, until a match is located:

If *S* contains a declaration of an explicit interface member implementation that matches *I* and *M*, then this member is the implementation of *I.M*.

Otherwise, if *S* contains a declaration of a non-static public member that matches *M*, then this member is the implementation of *I.M*.

An error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of *C*. For purposes of this process, the members of an interface include those members that are inherited from base interfaces.

For purposes of interface mapping, a class member *A* matches an interface member *B* when any one of the following is true: (1) *A* and *B* are methods, and the name, type, and formal parameter lists of *A* and *B* are identical, (2) *A* and *B* are properties, the name and type of *A* and *B* are identical, and *A* has the same accessors as *B*, wherein *A* is permitted to have additional accessors if it is not an explicit interface member implementation, (3) *A* and *B* are events, and the name and type of *A* and *B* are identical and (4) *A* and *B* are indexers, the type and formal parameter lists of *A* and *B* are identical, and *A* has the same accessors as *B*, wherein *A* is permitted to have additional accessors if it is not an explicit interface member implementation.

There are notable implications of the above interface mapping algorithm(s). First, explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member. Second, private, protected, and static members do not participate in interface mapping.

In exemplary code 500 of Fig. 5A, the *ICloneable.Clone* member of *C* becomes the implementation of *Clone* in *ICloneable* because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. In exemplary code 510 of Fig. 5B, the *Paint* methods of both *IControl* and *IForm* are mapped onto the *Paint* method in *Page*. It is also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For instance, an implementation of the interface of exemplary code 520 of Fig. 5C would require at least one explicit interface member implementation, and would take one of the forms illustrated by code 530 of Fig. 5D.

When a class implements multiple interfaces that have the same base interface, there

can be only one implementation of the base interface. In exemplary code 540 of Fig. 5E, it is not possible to have separate implementations for the *IControl* named in the base class list, the *IControl* inherited by *ITextBox*, and the *IControl* inherited by *IListBox*. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of *ITextBox* and *IListBox* share the same implementation of *IControl*, and *ComboBox* is simply considered to implement three interfaces, *IControl*, *ITextBox*, and *IListBox*.

As mentioned above, the members of a base class participate in interface mapping. In exemplary code 550 of Fig. 5F, the method *F* in *Class1* is used in *Class2*'s implementation of *Interface1*.

With respect to interface implementation inheritance in accordance with the present invention, a class inherits all interface implementations provided by its base classes. Without explicitly re-implementing an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations of exemplary code 600 of Fig. 6A, the *Paint* method in *TextBox* hides the *Paint* method in *Control*, but it does not alter the mapping of *Control.Paint* onto *IControl.Paint*. Calls to *Paint* through class instances and interface instances will have the effects illustrated by code 610 of Fig. 6B.

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the declarations of Fig. 6A to the declarations of code 620 of Fig. 6C, the effects of code 630 of Fig. 6D are observed.

Since explicit interface member implementations are not declared virtual, it is not possible to override an explicit interface member implementation. It is, however, perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. In exemplary code 640 of Fig. 6E, classes derived from *Control* can specialize the implementation of *IControl.Paint* by overriding the *PaintControl* method.

With respect to interface re-implementation in accordance with the present invention, a class that inherits an interface implementation is permitted to re-implement the interface by including it in the base class list. A re-implementation of an interface follows the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect on the interface mapping established for the re-

implementation of the interface. For example, in the declarations of exemplary code 700 of Fig. 7A, the fact that *Control* maps *IControl.Paint* onto *Control.IControl.Paint* doesn't affect the re-implementation in *MyControl*, which maps *IControl.Paint* onto *MyControl.Paint*.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. In exemplary code 710 of Fig. 7B, the implementation of *IMethods* in *Derived* maps the interface methods onto *Derived.F*, *Base.IMethods.G*, *Derived.IMethods.H* and *Base.I*. When a class implements an interface, it implicitly also implements all of the interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. In exemplary code 720 of Fig. 7C, the re-implementation of *IDerived* also re-implements *IBase*, mapping *IBase.F* onto *D.F*.

The invention of explicit interface member implementation provides a solution to a number of shortcomings in the prior art, namely the lack of ability to handle implementation of two independently-developed and conflicting interfaces, the lack of ability to privately implement an interface, and the lack of ability to specifically implement generic explicit interface members.

What follows is a more detailed description of these scenarios in order to understand both the drawbacks of interfaces that lack the benefits of the present invention, and the benefits that an explicit interface member implementation brings to a programming language, such as C#.

Interfacing - Implementing two independently-developed and conflicting interfaces

It is common for an interface to be developed by one party and to be implemented by another, and also for one class or struct to implement multiple interfaces. Given the independent nature of this development, conflicts sometimes arise. For instance, it is possible for two interface designers to select the same interface member name. One developer named FirstParty might deliver the interface represented by code 800 of Fig. 8A and another named SecondParty might deliver the interface represented by code 810 of Fig. 8B.

When the (1) signature, which may be based on name, parameter number, parameter types, (2) return value, and (3) contractual semantics of the methods are identical, this typically does not present a problem. C# and some other languages permit a class or struct to

implement both interface methods with a single method, such as is illustrated by the code 820 of Fig. 8C. However, given the fact that the interfaces were independently developed, this confluence of factors is extremely unlikely. In a particularly bad scenario, the methods have the same signature, but a different return value, as illustrated by code 830 of Fig. 8D.

5 In the scenario of Fig. 8D, it is not possible for a single class to implement both interfaces, since overloading based on return type is not permitted. This overloading restriction is common in object-oriented programming languages. The program 840 of Fig. 8E is also illegal. This limitation can present difficulties for developers, who generally are forced to workaround the problem by implementing the two interfaces on different objects.

10 This workaround is possible but inconvenient, makes developing more complex, and generally consumes time and computing resources.

Explicit interface member implementation addresses this scenario in a much better way. The class *C* can be fixed by implementing either or both of the interface members using explicit interface member implementation. The program 850 of Fig. 8F uses explicit interface member implementation in accordance with the invention for both *Interface1.F* and *Interface2.F*.

Interfacing - Privately implementing an interface

In other current programming languages, interface members can only be implemented using public members. In certain circumstances, it is convenient for a class to implement an interface, but in a private manner. By doing so, a class can both implement the interface and avoid cluttering up the class definition. This can be an important factor for class library providers because the consumers of class libraries typically look to the public members of a class for information on how to use the class.

25 For instance, the class *C* implemented by code 900 of Fig. 9 implements *Interface1* using private interface member implementation according to the present invention. A class consumer would look at *C* with an object browser or other tool and see that *C* implements the *Interface1* interface, and that *C* provides a *G* method. Note that the class consumer does *not* see an *F* method on *C*. The omission of this member provides value to the developer of *C*.

30 The lack of an *F* helps call attention to the method *G*, which is what consumers should actually be concerned with. The implementation of *Interface1.F* can still be called, but it is

necessary to cast to *Interface1* first.

Interfacing - Generic-specific programming

A specific example of privately implementing an interface relates to generic-specific programming, wherein a *generic* interface or group of interfaces is defined, and one or more implementers provide *specific* implementations. For instance, implementers might agree on a set of collection interfaces that are based on a generic type like object, as illustrated in code 1000 of Fig. 10A. The implementers might then provide specific implementations for strongly typed collections of various types. Without explicit interface member implementation, as illustrated by the code 1010 of Fig. 10B, it is not possible to provide a strongly typed member named “Current” since this name is already taken by the name of the member that implements *Ienumerator.Current*.

This lack of a strongly typed member has several negative effects including (1) performance during iteration is slowed down, as an explicit conversion from *object* to *Item* is required, (2) developer productivity is negatively impacted because developers typically forget to do the cast, and this leads to a compile-time error and (3) developer productivity is negatively impacted because development tools that are based on strong typing, e.g. statement completion, are hampered in this scenario.

The *ItemCollection* can be fixed by implementing *IEnumerator.Current* using explicit interface member implementation, thus keeping the generic implementation out of the public interface of *ItemCollection* and adding a strongly typed public member named *Current*. Typically, the generic member can simply defer to the strongly typed member, and no conversions are required, as illustrated by code 1020 of Fig. 10C.

As mentioned above, while exemplary embodiments of the present invention have been described in connection with the C# programming language, the underlying concepts may be applied to any programming language for which it would be desirable to have interfacing as described herein. Thus, an explicit interface member in accordance with the present invention may be implemented with any programming language, such as Fortran, Pascal, Visual Basic, C, C++, Java etc. In accordance with the present invention, a developer may utilize, or code, with a programming feature, namely an explicit interface member

mechanism as described above, that enables a software component to implement an explicit interface member by explicitly specifying the relationship between the software component and the interface member.

The various techniques described herein may be implemented in connection with hardware or software or, where appropriate, with a combination of both. Thus, the methods and apparatus of the present invention, or certain aspects or portions thereof, may take the form of program code (*i.e.*, instructions) embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other machine-readable storage medium, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. In the case of program code execution on programmable computers, the computing device will generally include a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device. One or more programs utilizing the interfaces of the present invention are preferably implemented in a high level procedural or object oriented programming language to communicate with a computer system. However, the program(s) can be implemented in assembly or machine language, if desired.

The methods and apparatus of the present invention may also be embodied in the form of program code that is transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via any other form of transmission, wherein, when the program code is received and loaded into and executed by a machine, such as an EPROM, a gate array, a programmable logic device (PLD), a client computer, a video recorder or the like, the machine becomes an apparatus for practicing the invention. When implemented on a general-purpose processor, the program code combines with the processor to provide a unique apparatus that operates to include the interfacing functionality of the present invention. For example, any storage techniques used in connection with the present invention may invariably be a combination of hardware and software.

While the present invention has been described in connection with the preferred embodiments of the various figures, it is to be understood that other similar embodiments may be used or modifications and additions may be made to the described embodiment for performing the same function of the present invention without deviating therefrom. For

example, while exemplary embodiments of the invention are described in the context of programming in a networked or .NET computing environment, one skilled in the art will recognize that the present invention is not limited thereto, and that the methods of programming in a programming environment having explicitly defined interface members, as
5 described in the present application may apply to any computing device or environment, such as a gaming console, handheld computer, portable computer, etc., whether wired or wireless, and may be applied to any number of such computing devices connected via a communications network, and interacting across the network. Furthermore, it should be emphasized that a variety of computer platforms, including handheld device operating
10 systems and other application specific operating systems are contemplated, especially as the number of wireless networked devices continues to proliferate. Therefore, the present invention should not be limited to any single embodiment, but rather should be construed in breadth and scope in accordance with the appended claims.

105020 "EEF00660